Programmation coté serveur

M. PERREIRA DA SILVA

Version PDF des slides

Rôle d'un serveur web

- **Génération** de document
  - HTML, XML, etc.
- **Accès** aux données
  - Fichiers (faible volume de données)
  - SGBD (gros volume de données)
- **Concurrence**
  - Ordonnancement des connexions, entrées-sorties, etc.
- **Sécurité**
  - Restrictions d'accès aux fichiers présents sur le serveur
- **Sessions**
  - Maintenir une conversation cohérente avec le client

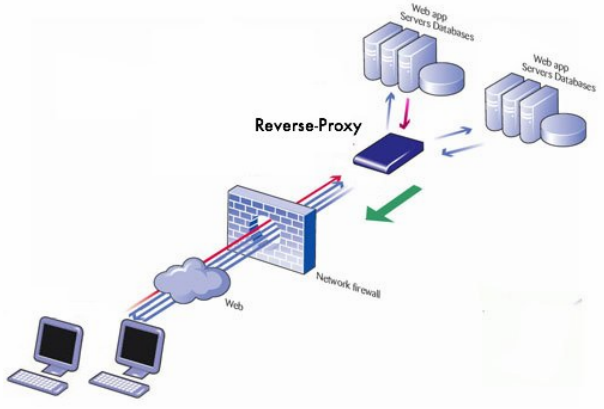
Serveurs web généralistes

- **Apache**: serveur open source de référence (40% des sites*). Existe depuis 1995
- **IIS**: Le serveur Web fournit par Microsoft pour la plateforme NT (27% des sites). Seule solution pour faire fonctionner les solutions Microsoft (ASP / ASP.Net). Existe depuis 1994
- **Nginx**: serveur **asynchrone** open source Russe (15% des sites). Existe depuis 2002
- **Google Web Server**: version modifiée d'Apache. Code non public, uniquement utilisé par Google. (2% des sites)

* Source: étude Netcraft janvier 2015

Proxy inverse

- Proxy web: donne accès a internet à partir d'un réseau LAN
- Proxy inverse: donne accès à un réseau LAN à partir d'internet
  - **Cache** pour décharger les serveurs webs d'une partie de leur travail
  - **Filtrage** des accès aux ressources web depuis l'extérieur
  - **Chiffrement** des connexions
  - **Répartition** de charge entre plusieurs serveurs
  - **Compression**
  - **Mutualisation** de plusieurs serveurs web sur une même machine / adresse



Contrôle d'accès

Exemple: .htaccess d'Apache

- **Chaque répertoire** peut avoir son fichier `.htaccess`
- On peut **autoriser / interdire** l'accès

```
order deny,allow
allow from univ-nantes.fr # l'accès à partir de l'université est permis
deny from all             # mais est interdit pour les autres
```

- On peut **protéger** un répertoire par **mot de passe**

```
AuthType Basic  # Authentification basic HTTP (peu sécurisée)
AuthUserFile /users/mperreir/www/repertoire/.passwds # endroit où vous conservez
les mots de passe
AuthName "Entrez votre mot de passe" # ce qui figurera dans la barre de titre de
la fenêtre d'autentification
require valid-user # n'importe quel utilisateur de .passwds est accepté
```

Hôtes virtuels

- Un serveur web peut héberger **plusieurs sites web**, chacun à partir d'une arborescence spécifique
  - Ex: http://www.site1.fr à partir de /var/www/site1
  - Ex: http://www.trucbidule.com à partir de var/www/trucbidule
- Chaque hôte virtuel à sa **configuration propre**
- Les ressources du serveur sont **partagées** par les différents hôtes
- Le serveur web **différencie** les hôtes par (au choix)
  - Nom d'hôte
  - Adresse IP
  - Port (rarement utilisé)

Réécriture d'URL

- Accès à une ressource web effectué généralement via une requête `GET`
  - Ex: http://www.notre-site.com/articles/article.php?id=12&page=2&rubrique=5
- **Peu lisible**, et permet de connaitre la technologies (PHP) et les variables utilisées (`id`, `page`, `rubrique`). On préfèrerait:
  - http://www.notre-site.com/articles/article-12-2-5.html
- La solution: **réécrire** les URL
  - Exemple (Apache): via des **expressions régulières**

```
RewriteEngine On  # activation de la réécriture d'URL
# On définit une Regex qui transforme la "jolie" URL en l'URL réellement
interprétée par le serveur
RewriteRule ^/articles/article-([0-9]+)-([0-9]+)-([0-9]+)\.html$
/articles/article.php?id=$1&page=$2&rubrique=$3 [L]
```

Serveur web et scripts

- **Site statique**: le serveur web renvoie directement la ressource (fichier) demandée
- **Site dynamique**: un programme / script est exécuté afin de **générer** la ressource demandée (ex: page HTML)
  - Langage utilisable: **tout langage** pouvant générer du texte
  - Mais certains langages sont plus adaptés
    - Librairies disponibles
    - Facilité à manipuler du texte
  - Deux stratégies
    - **génération de code** HTML (CGI, WSGI, Servlets, etc.)
    - **préprocesseurs** HTML (PHP, ASP, JSP, etc)

Technologies basées sur la génération de code

CGI, WSGI, Servlets, etc.

CGI

- CGI (Common Gateway Interface) : interface **normalisée** permettant de faire communiquer le serveur Web avec un programme s'exécutant sur le serveur
- On peut utiliser **n'importe quel langage**
  - Compilé, comme C, C++, Java
  - Interprété, comme Perl, Python, Ruby, etc.
- Un programme CGI communique avec le serveur via:
  - Les **variables d'environnement**: lecture des entêtes HTTP de la requête
  - Le **flux standard** d'entrée: lecture des données de la requête (ex: requête POST)
  - Le **flux standard** de sortie: écriture de la réponse (entêtes HTTP + données)
- Le programme CGI est appelé par le serveur web à **chaque requête**

CGI: limites

- Avantages CGI
  - **Simplicité**
  - Indépendance par rapport aux langages de programmation
- Inconvénients CGI:
  - Simplicité
  - **Sécurité**
  - **Performances** (1 requête = 1 processus lancé)
    - Problème réglé par **FastCGI** et **SCGI**

CGI: exemple

Un simple **script shell**

```
#!/bin/sh
# exemple.cgi

# Génération des entêtes HTTP
echo "Content-type: text/html"
echo  #ligne vide pour signaler la fin des entêtes

# Creation du corps du document (on a omis le doctype ici)
echo "<html><head><title>Exemple.cgi</title></head>"
echo "<body>"
echo "<h1>Bonjour !</h1>"

# Les paramètres passés dans l'URL (query string) sont lus à partir d'une
variable d'environnement
echo "Voici la chaine de requête qui m'a été passée=$QUERY_STRING"

# Les données de formulaire (requête POST) ou le contenu du fichier (requête PUT)
sont passées sur l'entrée standard
read DATA
echo "Et ici ce sont les données (POST)=$DATA"
echo "</body></html>"
```

Bilan: CGI, Servlets, WSGI, etc.

- **Avantages**
  - On écrit un programme "presque classique"
- **Inconvénients**
  - On doit généralement générer du code HTML...
  - Le code HTML n'est pas séparé du code serveur
    - Ex (Java): `System.out.println("<h1>Ceci est un titre</h1");`
  - Maintenabilité du code HTML complexe !
  - Besoin de séparer code HTML et code serveur

Préprocesseurs HTML

PHP, ASP(.Net), JSP, etc.

"Préprocesseur" HTML

- **Rôle prépondérant** de HTML
  - Pages écrites en HTML
  - Ajout (minimal) de scripts dans le HTML
  - Code métier dans des fichiers externes
- Différence avec code JavaScript intégré au HTML : **exécution sur le serveur** !
  - L'utilisateur ne voit pas et ne peut pas modifier le code...
- **Avantages**
  - Meilleure séparation "présentation" / "code métier"
  - Meilleure lisibilité / maintenabilité
  - Simplification du code à écrire (moins de headers, etc.)
- **Inconvénients**
  - Utile seulement si on doit générer des pages HTML...

PHP (1/2)

- Langage **interprété**, **faiblement typé**, créé en 1994 par Rasmus Lerdorf
- Version actuelle : 7.4 (version 5.x de 2004 à fin 2018!)
- Doit être **intégré à un serveur web** (ex: Apache via mod_php)
- Un fichier `.php` est traité par PHP comme
  - un fichier HTML
  - avec des **balises spéciales** pour
    - Exécuter du code

```php
<?php echo 'Hello ', 'World'; ?>
```

    - Insérer la valeur d'une constante / variable dans le code HTML

```php
<?= 'Hello World' ?>
```

PHP (2/2)

- La syntaxe du langage est **proche de celle du C**
- API et bibliothèques externes très riches du fait de sa **large utilisation**
- Accès aux entêtes HTTP et données via des **tableaux associatifs** (super)globaux
  - `$_GET` : données de la chaine de requête (GET)
  - `$_POST` : données de formulaire (POST)
  - `$_REQUEST` : concaténation des données de `$_GET` et `$_POST`
  - `$_FILES` : informations sur les fichiers envoyés via une requête `$_POST`
  - `$_SESSION` : données stockées dans une **session**
  - `$_COOKIE` : les cookies envoyés par le client
  - `$_SERVER` : entêtes HTTP et autres données passées par le serveur web
  - `$_ENV` : les variables d'environnement (dont les variables CGI)

PHP: exemple

mon_form.html

```
<html>
    <body>
        <form action="bonjour.php" method="post">
            <p>Nom: <input type="text" name="nom"></p>
            <p>Mail: <input type="email" name="email"></p>
            <input type="submit">
        </form>
    </body>
</html>
```

bonjour.php

```
<html>
    <body>
        Bonjour <?php echo $_POST["name"]; ?><br>
        Ton adresse mail est: <?= $_POST["email"] ?><br>
        Ton navigateur est: <?php echo $_SERVER["HTTP_USER_AGENT"] ?>
    </body>
</html>
```

PHP: exemple

On peut aussi écrire des programmes PHP qui **génèrent du code** HTML (à la Servelet, CGI, etc.)

bonjour.php

```php
// init de la chaine qui contiendra le code HTML généré
$out = "";

// On génère le code de la réponse à la requête POST
$out += "<html>";
$out += "<body>";
$out += "Bonjour " . $_POST["name"] . "<br>";
$out += "Ton adresse mail est: " . $_POST["email"] . "<br>";
$out += "Ton navigateur est: " . $_SERVER["HTTP_USER_AGENT"];
$out += "</body>";
$out += "</html>";

// on renvoie le code HTML au serveur web
echo $out;
```

Technologie sans serveur web externe

NodeJS

NodeJS : c'est quoi ?

- Outils de création d'application web avec du **code javascript coté serveur**
- Utilise le moteur JavaScript "V8" de Google
  - **Rapide** !
- NodeJS n'est **PAS**:
  - Un framework Web (mais il en existe pour NodeJS)
  - De haut niveau
    - Pratique pour voir les bases des technologies web
    - Il existe de nombreux modules pour simplifier le développent
  - Multi-threadé
    - **Une seule instance** de votre code est exécutée

NodeJS est également un serveur web...

- Un programme NodeJS fait tourner **son propre serveur** web via l'objet http
  - Les réponses aux requêtes se font via une **boucle d'évènements**
  - Une fonction est appelée à chaque évènement
- Exemple:

serveur.js

```
var http = require('http'); // import du module http
var server = http.createServer(function(request, response) {
    response.writeHead(200); // code de statut HTTP
    response.write("Bonjour tout le monde !"); // contenu de la réponse
    response.end(); // on envoie la réponse
}).listen(8080);
console.log('Serveur lancé sur le port 8080...');
```

- On lance le programme (serveur) avec: `node serveur.js`

Gestion des requêtes et des réponses

- `http.createServer` prend en paramètre une fonction à deux arguments (`request` et `response`)
  - `request` est de type `http.IncomingMessage`
    - accès aux **entêtes HTTP** via le tableau associatif `headers`
  - `response` est de type `http.ServerResponse`
    - on écrit les **entêtes** avec la méthode `writeHead` (à appeler avant `write`)
    - on écrit le **corps de la réponse** avec la méthode `write` (en une ou plusieurs fois)
    - on **termine et envoie** la réponse avec un appel à `end`

Gestion des évènements

- De nombreux objets de Node.JS sont des instances de la classe `EventEmitter`
- Ils possèdent une méthode `on(event, listener)` qui permet de déclarer quel callback sera appelé pour quel évènement

```
server.on('connection', function (stream) {
  console.log('someone connected!');
});
```

Nodejs: modules

- NodeJS est extensible via des **modules**, pouvant être écrits par n'importe quel développeur
- NPM (Node Package Manager) est le **gestionnaire de modules** intégré à NodeJS
  - Installation d'un nouveau module `npm install nom_du_module`
  - Importation d'un module dans une application `var module = require('nom_du_module')`
- Toutes les **fonctionnalités internes** de NodeJS doivent être accédées via le mécanisme des modules
  - Gestion du protocole HTTP: `var http = require('http');`
  - Accès au système de fichier du serveur: `var fs = require('fs');`
  - Manipulation des chaines de requête (query string): `var querystring = require('querystring');`
  - Etc.

Nodejs: frameworks

- NodeJS propose des **fonctionnalités de bas niveau**
  - Besoin d'outils d'un peu plus haut niveau pour être productif
- Les **frameworks** web répondent à cette problématique (dans tous les langages web)
- **ExpressJS** est un framework web léger (minimaliste ?) pour NodeJS*
- Fonctionnalités:
  - Routage: quelle fonction est associée à quelle URL ?
  - Automatisation du traitement de certaines requêtes via des "Middleware"
  - Intégration de différents moteurs de "Template"
  - Gestion des erreurs
  - Générateur (de squelette) d'application web
- ExpressJS est un **module** de NodeJS : `var express = require('express');`

* Il existe bien entendu d'autres frameworks pour NodeJS (ex: Koa)

ExpressJS: exemple

```
var express = require('express'); // on charge ExpressJS
var app = express();    // on récupère notre application

// routage : une requête GET effectuée à la racine du site
// déclenchera cette fonction
app.get('/', function (req, res) {
  res.send('Bonjour !');
});

// Création du serveur web pour notre application sur le port 8080
var server = app.listen(8080, function () {

  var host = server.address().address;
  var port = server.address().port;

  console.log('Application lancée à l\'adresse suivante http://%s:%s', host,
port);

});
```

ExpressJS: routing

- Une **route** permet de définir quelles fonctions (callback) seront exécutées pour une **URL** et une **méthode HTTP** (GET, POST, etc.) données
- Syntaxe: `app.METHODE( url, [callback...], callback)`
  - L'URL peut être une chaine de caractères ou une expression régulière
  - Si on précise plusieurs callbacks, chaque callback doit appeler le suivant via la fonction `next()`

```
app.post('/example/b', function (req, res, next) {
    console.log('La réponse sera renvoyée par la fonction suivante ...');
    next();
}, function (req, res) {
    res.send('Vous êtes dans B!');
});

// fonctionnera pour "polytech", "polyjoule", "polyson", etc.
app.get(/^poly.*/, function(req, res) {
  res.send('Poly quelque chose !');
});
```

ExpressJS: requête et réponse

- Objet `request` (1er paramètre du callback d'une route)
  - Accès aux **entêtes HTTP** (Ex: `request.body`, `request.cookies`, etc.)
  - Nécessité d'un module externe (`body-parser`) pour décoder le **corps des requêtes** POST
- Objet `response` (2nd paramètre du callback d'une route)
  - `response.sendStatus()`: envoi du code de statut et du message correspondant
  - `response.send()`: envoi de données (chaine, objet, tableau, etc.)
  - `response.sendFile()`: envoi d'un fichier binaire
  - `response.redirect()`: demande au client d'effectuer une redirection
  - `response.render()`: effectue le rendu d'un "template"
  - `response.end()`: signale la fin de la réponse
  - Etc.

ExpressJS: exemple 2

On utilise le même fichier HTML 'mon_form.html' que précédemment

```
var app = require('express')();
var bodyParser = require('body-parser');

// Pour décoder un formulaire encodé avec 'application/x-www-form-urlencoded'
app.use(bodyParser.urlencoded({ extended: true }));

// Une route pour gérer les requêtes POST à la racine du site
app.post('/', function (req, res) {
    res.send('<html><body>');
    res.send('Bonjour ' + req.body.nom + '<br>');
    res.send('Ton adresse mail est: ' + req.body.email + '<br>');
    res.send('Ton navigateur est: ' + req.get('User-Agent'));
    res.send('</body></html>');
})

// on lance le serveur web (silencieusement) sur le port 8080
app.listen(8080);
```

Nodejs: template (jade, ejs, etc.)

- On aimerait pouvoir séparer code HTML du code JavaScript (comme en PHP, JSP)...
- C'est le rôle des **moteurs de template**:
  - Jade: permet de générer du HTML à partir d'un dialecte allégé (ex: `h1` à la place de `<h1>`), mais paramétrable (boucles, variables, etc.)
  - EJS: Extension de HTML pour permettre d'enrichir le code HTML via du code JavaScript. Syntaxe proche de celle de JSP
  - Et bien d'autres (Hogan.js, DoT.js, **Mustache.js**, Handlebars.js, etc.)
- On peut utiliser les templates avec NodeJS seul ou avec NodeJS+ExpressJS (plus simple)

ExpressJS + EJS

Partie template EJS

users.html

```
<html>
<head>
  <meta charset="utf-8">
    <title> <%= title %> </title>
  </meta>
</head>
<body>
<h1>Utilisateurs</h1>
    <ul id="users">
    <% users.forEach(function(user){ %>
        <li><%= user.name %> &lt;<%= user.email %>&gt;</li>
    <% }) %>
</ul>
</body>
</html>
```

ExpressJS + EJS

Partie code serveur NodeJS / ExpressJS

server.js

```
var express = require('express');
var app = express();

// On charge et on déclare EJS comme moteur de template
app.engine('.html', require('ejs').__express);

// Nos utilisateurs
var users = [
  { name: 'pierre', email: 'pierre@polytech.fr' },
  { name: 'paul', email: 'paul@polytech.fr' },
  { name: 'jacques', email: 'jacques@polytech.fr' }
];

// On affiche les utilisateur lors d'une requête GET à la racine du site
app.get('/', function(req, res){
  res.render('users.html', {
    users: users,
    title: "Exemple d'utilisation d'EJS"
  });
});

app.listen(8080);
```

Client side frameworks

AngularJS

- Framework développé par Google pour le développement d'applications "Single Page"
- **Déporte** une grande partie des actions généralement effectuées sur le serveur **vers le client**
  - Moteur de templates
  - Mise à jour du HTML en fonction des données
  - Navigation dans l'application
- Le **serveur** n'est plus chargé que de vérifier, valider et envoyer les **données**
- Basé sur le design pattern **Modèle Vue Contrôleur**
  - Modèle: Données sur le serveur, envoyées en JSON ou XML
  - Vue: Le code HTML mis à jour par AngularJS
  - Contrôleur: Code JavaScript permettant de mettre à jour la vue en fonction des données du modèle

Nombreux autres frameworks : Vuejs, Reactjs, Angular, Svelte, etc.

AngularJS: exemple

index.html

```
<html>
  <head>
    <title>My Angular App!</title>
    <script src="angular.min.js"></script>
    <script src="app.js"></script>
  </head>
  <body ng-app="monAppAngular" ng-controller="MainCtrl">
    <div> {{test}} </div>
    <div ng-repeat="elt in liste | orderBy: '+nom'">
        {{elt.nom}} - {{elt.desc}}
    </div>
  </body>
</html>
```

app.js

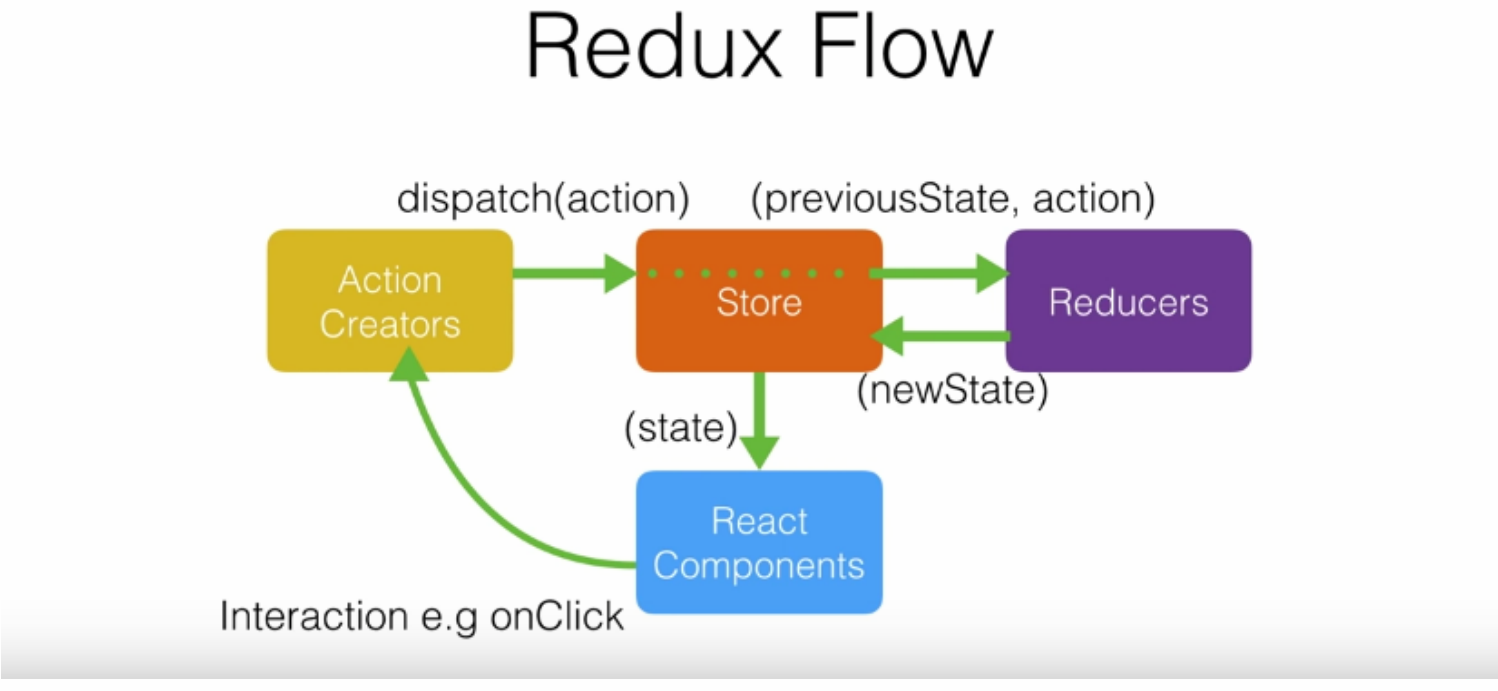
```
var app = angular.module('monAppAngular', []);

app.controller('MainCtrl', [
'$scope',
function($scope){
 $scope.test = 'Hello world!';
 $scope.liste = [{nom: 'e1', desc: 'element 1'}, {nom: 'e2', desc: 'element 2'}];
}]);
```

Client side frameworks

Gestion des états

- Problème de passage à l'échelle des applications coté client
- Comment gérer l'état de tous les composants (pure HTML ou ReactJS) ?
- Flux / Redux => notion de store, d'état et d'action



Répartition du code : client vs serveur

- Ce qui est **toujours coté serveur** = le code "métier"
  - Accès et manipulation des données
  - Règles "métier" de traitement des données
  - Accessible directement ou via un service web
- Ce qui est **toujours coté client** (navigateur) = interactions
  - Réaction aux évenement d'interface utilisateur (clics, clavier, etc.)
- Tout le reste peut être **coté client ou serveur**
  - Routing
  - Gestion des états
  - Génération des pages (templates)

Plus loin avec NodeJS

- Un cours bien fait: http://courseware.codeschool.com/node_slides.pdf
- La doc de NodeJS: https://nodejs.org/api/
- La doc d'ExpressJS: http://expressjs.com/
- Les tutoriels NodeSchool: http://nodeschool.io/
  - En particulier learnyounode
- Un tutoriel MEAN: https://thinkster.io/mean-stack-tutorial/

Prochainement: les services web